kansari-midterm

March 16, 2024

1 ENPM673 Perception for Autonomous Robots (Spring 2024) - Midterm Exam

1.0.1 Deadline: 16th March 7AM

Rohan Maan, Jay Prajapati, Samer Charifa, Tommy Chang

1.1 Submission guidelines

- The midterm exam is to be done and submitted individually.
- Write the solution to **all the problems in one Google Colab File. (Handwritten answers will not be allowed)**
- Your submission on ELMS/Canvas should be the following:
  a. Google Colab (.ipynb) file
  b. Google Colab (.pdf) file (Convert the same .ipynb file to a .pdf file) following the naming convention YourDirectoryID_midterm.
- Points will be deducted if you don't follow the submission guidelines.
- Submit all files in one attempt.
- Provide detailed explanations for each step using text cells in Google Colab.
- Ensure the code is well-formatted for readability.
- Comment on each section (preferably every 2 lines) of the code to explain its purpose and functionality.
- Include relevant output images within the text cells where required.
- Note that the use of ChatGPT (or any other generative AI websites) is not allowed.
- You are free to use any in-built OpenCv functions **except for PCA (In Question-3)**.

1.2 How to use this file

- Links to all the input data have been given in the text cells.
- Add this .ipynb file to your working directory on google drive.
- Add the given input files to your working directory on google drive.
- Make sure you are using Google Colab and not any other IDE.
- Placeholders to write the answers have been given in this file.
- Do not change the structure of the questions.
- You may add extra code cells or text cells for any particular question.
- Make sure that the extra cell is added under the right question.

1.3 Enter Student Details Here

Name	Kashif Ansari
UID	120278280
Email	kansari@umd.edu

2 Set path to the working directory

[2]:
```
#Connecting google drive
from google.colab import drive
drive.mount('/content/drive/', force_remount=True)
```

```
Mounted at /content/drive/
```

[3]:
```
# Change the working directory
path_to_folder = "ENPM673/tutorials"
%cd /content/drive/My\ Drive/{path_to_folder}
```

```
/content/drive/My Drive/ENPM673/tutorials
```

3 Import Libraries

[4]:
```
# Importing libraries

import cv2
import numpy as np
import random
import matplotlib.pyplot as plt
import csv
import plotly.graph_objects as go
import pandas as pd
from numpy.linalg import lstsq
```

4 Problem 1 (20 Points)

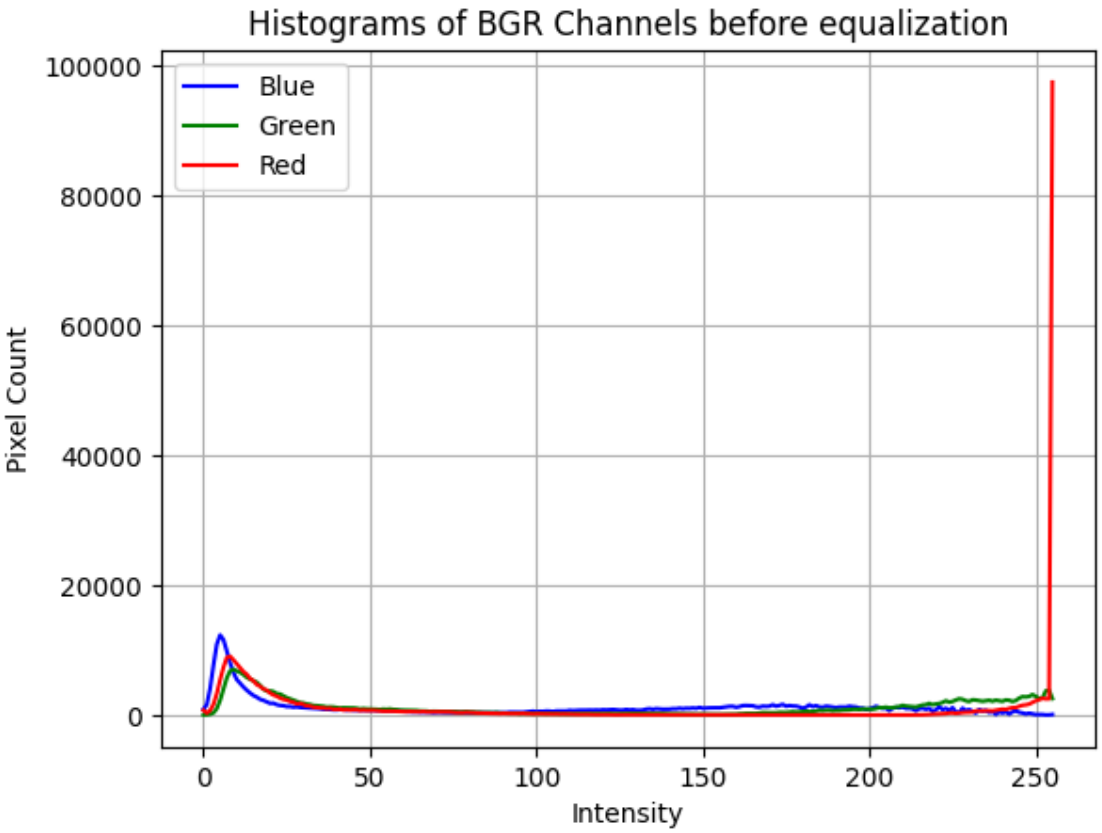
Link to input image: https://drive.google.com/file/d/1X0xir8CrOoKsNXxvZnLVAJWKztpMuUhV/view?usp=dri

Part 1: Write code and describe the steps to perform histogram equalization of the given color image. Hint: First convert to a color space that separates the intensity from the color channels (eg. LAB or LUV color spaces work the best). Your result should look like the below: Link to the output of part-1: https://drive.google.com/file/d/1nIhoE0PyCdFE3LfAwGcUqUt_GaQ2NoG-/view?usp=sharing

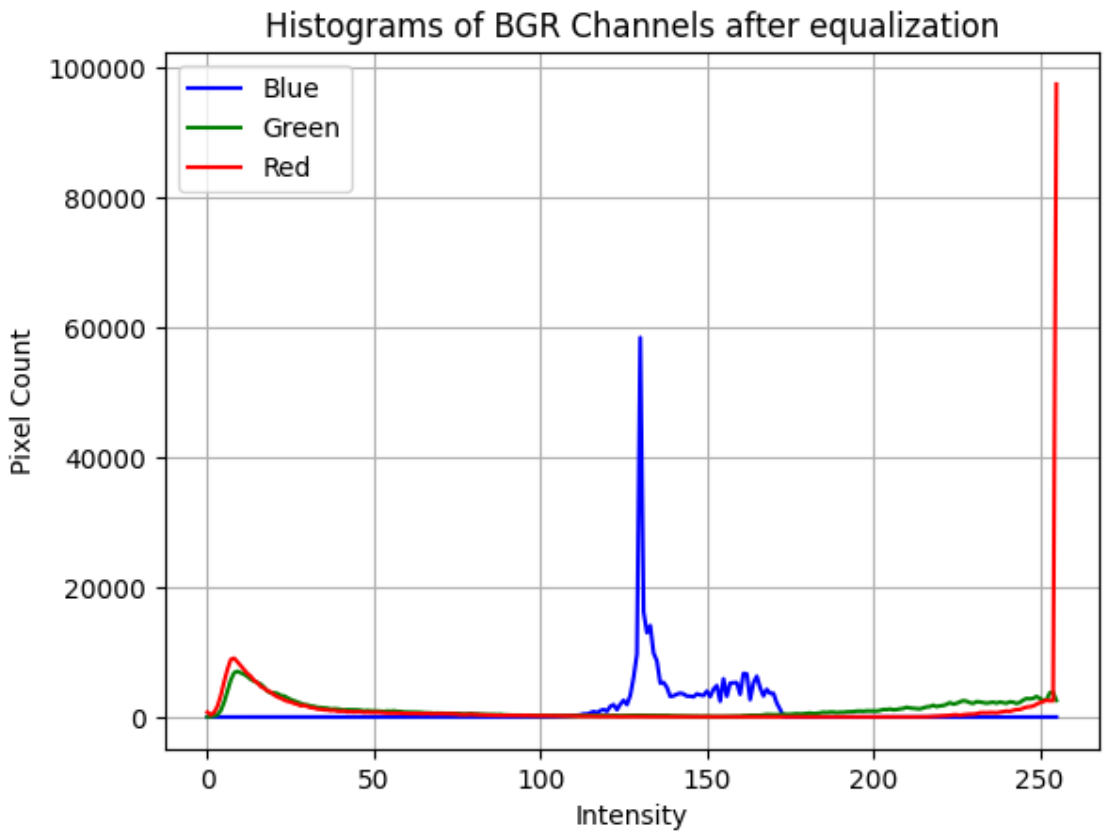
[5]:
```
# Fetching image
img_h = cv2.imread('assets/PA120272.JPG')
img_rgb = (cv2.cvtColor(img_h, cv2.COLOR_BGR2RGB))
plt.imshow(img_rgb)
```

```
plt.axis('off')  # Turn off axis labels
plt.title('Given Image')
plt.show()

#Splitting into different channels
b,g,r = cv2.split(img_rgb)

#Projecting the BGR channels as histogram before equalization
hist_b = cv2.calcHist([b], [0], None, [256], [0, 256])
plt.plot(hist_b, color='blue', label='Blue')

hist_g = cv2.calcHist([g], [0], None, [256], [0, 256])
plt.plot(hist_g, color='green', label='Green')

hist_r = cv2.calcHist([r], [0], None, [256], [0, 256])
plt.plot(hist_r, color='red', label='Red')

#Displaying the plot
plt.xlabel('Intensity')
plt.ylabel('Pixel Count')
plt.title('Histograms of BGR Channels before equalization')
plt.legend()
plt.grid(True)
plt.show()

# Converting the image into LAB space
image_lab = cv2.cvtColor(img_rgb, cv2.COLOR_BGR2LAB)

# Splitting the image into l(brightness) a(red-green) b(blue-yellow)
l,a,b = cv2.split(image_lab)

# Equalizing the l(brightness)
l = cv2.equalizeHist(l)

#Combining the splitted channels into one and converting back into original BGR␣
 ↪channels.
merge_lab = cv2.merge((l,a,b))
final_output = (cv2.cvtColor(merge_lab, cv2.COLOR_LAB2BGR))

#Displaying the output image
plt.imshow(final_output)
plt.axis('off')  # Turn off axis labels
plt.title('final_output Image')
plt.show()

#Projecting the BGR channels as histogram after equalization
hist_b = cv2.calcHist([b], [0], None, [256], [0, 256])
```

```
plt.plot(hist_b, color='blue', label='Blue')

hist_g = cv2.calcHist([g], [0], None, [256], [0, 256])
plt.plot(hist_g, color='green', label='Green')

hist_r = cv2.calcHist([r], [0], None, [256], [0, 256])
plt.plot(hist_r, color='red', label='Red')

#Displaying the plot
plt.xlabel('Intensity')
plt.ylabel('Pixel Count')
plt.title('Histograms of BGR Channels after equalization')
plt.legend()
plt.grid(True)
plt.show()
```


Histograms of BGR Channels before equalization
Blue
Green
Red
Pixel Count
Intensity
100000
80000
60000
40000
20000
0
0
50
100
150
200
250

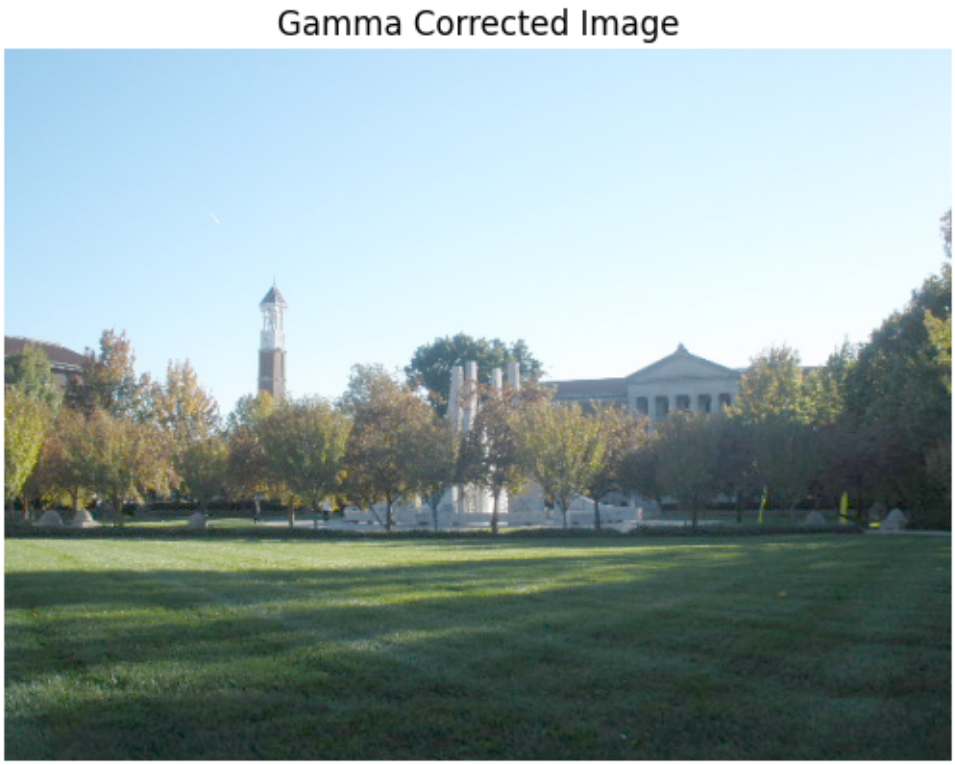
final_output Image



Part 2: Another way to enhance a dark image is to perform gamma correction. For each intensity pixel, replace it with the new intensity = 255*((intensity/255)^(1/2.2)). Write code to perform gamma correction. Your result should look like below: Link to the output of part-2: https://drive.google.com/file/d/1rgo7Kl8qK7Byh5QZsIb4CrfdBzY51Aco/view?usp=sharing

[6]:
```
# Read the image
img_h = cv2.imread('assets/PA120272.JPG')

# Convert BGR image to RGB
img_rgb = cv2.cvtColor(img_h, cv2.COLOR_BGR2RGB)

# Display the original image
plt.imshow(img_rgb)
plt.axis('off')
plt.title('Given Image')
plt.show()

# Gamma correction function
def gamma_correction(image, gamma):
```

```
    # Normalize pixel values to range [0, 1]
    normalized_image = image / 255.0
    # Apply gamma correction
    corrected_image = np.power(normalized_image, 1 / gamma)
    # Scale pixel values back to range [0, 255]
    corrected_image = (corrected_image * 255).astype(np.uint8)
    return corrected_image

# Performing gamma correction with gamma = 2.2
gamma = 2.2
img_corrected = gamma_correction(img_rgb, gamma)

# Display the corrected image
plt.imshow(img_corrected)
plt.axis('off')
plt.title('Gamma Corrected Image')
plt.show()
```



5 Problem 2 (15 Points)

1. For an image that is n by n and assuming that we have a random kernel call it H, where H is m by m matrix, what is the big O notation when performing convolution (how many multiplications are needed, in terms of n)?

2. Describe the meaning of "separable" kernel? Redo the previous question knowing that the H kernel is separable?

3. Apply the principle of separability to the following kernel to separate it into two kernels. Kernel

Problem-2 (1)

The number of multiplications needed for a convolution on an image of size (n x n) with a kernel 'H' of size (m x m) can be expressed in terms of 'Big O Notation'.

Denoting the size of the image as (n) and the size of the kernel as (m) while performing a 2D convolution, each output pixel requires (m x m) multiplications. Since there are (n x n) output pixels in total, the total number of multiplications can be calculated as (n x n) times (m x m). This can be explained as the Kernal being iterated for (m) rows and (m) columns Similarly for the image each pixel will pass through the loop/iteration for (n) rows and (n) columns

Hence, the Big O Notation for the number of multiplications needed to perform a convolution is O(n^2 x m^2).

Problem-2 (2)

A separable kernel refers to a 2D kernel that can be factored into two 1D kernels, one for its row and one for the corresponding column. This allows for more efficient computation during convolution as the operation can be broken down into two simpler 1D convolutions and the time complexdity reduces.

In this case for an image of size (n x n) and a separable kernel H (where H is (m x m)), the number of multiplications needed for convolution reduces significantly due to the separability property.

When a separable kernel is used for the given operation above, the convolution operation can be performed in two steps: first, convolve the image with the row kernel, and then convolve the result with the column kernel. Each of these 1D convolutions requires O(n x m) multiplications. Therefore, the total number of multiplications for separable convolution is O(2 x (n x n) x m) = O((n^2) x m).

Problem-2 (3) Given kernal = K = 1/16 [[1,2,1], [2,4,2], [1, 2, 1]]

In the above equation the sum of all the elements in the kernal is 16, Therefore the multiplying factor is (1/16)

Similarly in order to get the sum of all the elements in 1D kernal as '1' we multiply each element by (1/4).

This is given as:

K_normalized = (Each element of K) / (Sum of each element of K)

Hence K = K_row * K_col = ((1/4) x [1, 2, 1]) x ((1/4) [[1], [2], [1]])

[41]:
```
#Applying the principle of separability in python

# Given kernel
K = np.array([[1, 2, 1],
              [2, 4, 2],
              [1, 2, 1]])

# Factorize into row and column kernels
K_row = np.array([1, 2, 1]) / 4  # Normalize by 4
K_col = np.array([[1], [2], [1]]) / 4  # Normalize by 4

#printing seperated kernals
print("row kernal", K_row)
print("col kernal", K_col)
```

```
row kernal [0.25 0.5  0.25]
col kernal [[0.25]
```

```
[0.5 ]
[0.25]]
```

6 Problem 3 (20 Points)

Link to the csv file: https://drive.google.com/file/d/1LGG32bpU0sTIp-lOxOXCZJELGoZqVaZk/view?usp=sharing

Given x, y, z coordinates of n number of data points(problem3_points.csv). Consider matrix A where matrix A represents the coordinates of the n points shown in the figure below. Let's say we want to fit these data points onto an ellipse.

1. Describe the steps to find the major and minor axis of this ellipse along with their prospective length.

2. Similarly, given the data that represents a flat ground in front of the robot in 3D, try to extract the equation of the surface that describe the ground.

Problem-3 (1)

In order to fit a set of data points onto an ellipse and determine the major and minor axes along with their lengths. The data consisted of x, y, z coordinates of n data points obtained from the "problem3_points.csv" file. To achieve this, I first projectED the three-dimensional data points onto a 2D plane (x, y) for simplicity. I then utilized a linear least squares method implemented in the provided code to fit an ellipse to the data points, using the formula Y = BX, where Y is the vector of ellipse parameters, X is the design matrix composed of x and y coordinates, and B is the coefficient vector representing the ellipse equation. This process yielded coefficients (B) representing the ellipse equation. By analyzing the eigenvalues corresponding to these coefficients using the singular value decomposition (SVD) method, we identified the major and minor axes of the ellipse. The major axis length was determined by the square root of the largest eigenvalue, while the minor axis length was computed using the square root of the smallest eigenvalue. Additionally, the code allowed us to evaluate the ellipse equation Y = BX for visualization purposes.

[16]:
```
#Following definitions are created as per the above discussed methedology <<<


def fit_ellipse(x_data, y_data):
    # Construct the coefficient matrix
    X_matrix = np.column_stack((x_data * 2, x_data * y_data, y_data * 2,␣
 ↪x_data, y_data, np.ones_like(x_data)))

    # Solve the linear least squares problem
    _, _, V_matrix = np.linalg.svd(X_matrix)
    smallest_singular_value_index = np.argmin(np.abs(V_matrix))
    B_vector = V_matrix[smallest_singular_value_index, :]
```

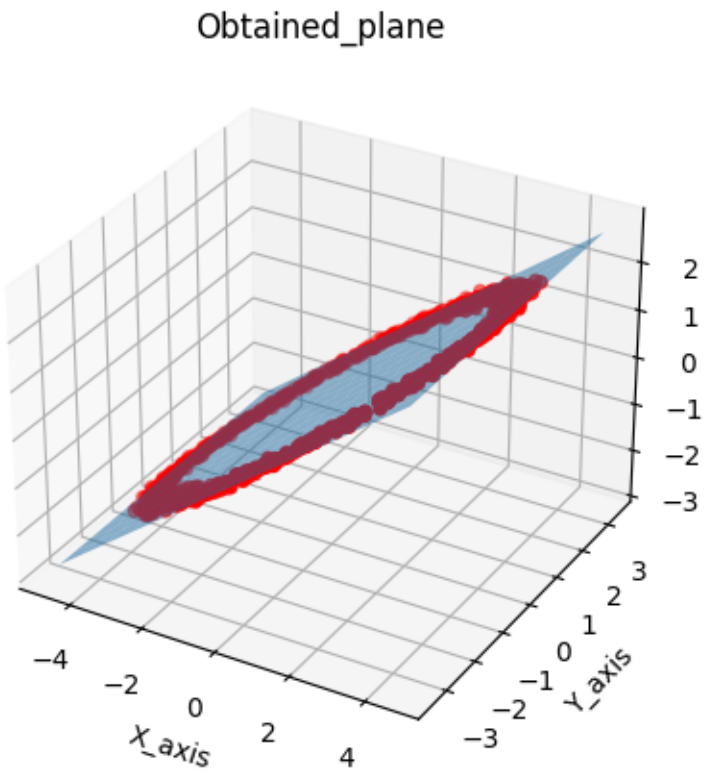
```
    return B_vector

def ellipse_equation(B_vector, x_data, y_data):
    # Evaluate the ellipse equation Y = BX
    Y_vector = np.dot(np.column_stack((x_data * 2, x_data * y_data, y_data * 2,␣
 ↪x_data, y_data, np.ones_like(x_data))), B_vector)
    return Y_vector
```

Problem-3 (2)

[86]:
```
# Reading CSV file
points_data = np.loadtxt("assets/problem3_points.csv", delimiter=",",␣
 ↪skiprows=1)

# Feeding the coordinates
X_coordi = points_data[:, 0]
Y_coordi = points_data[:, 1]
Z_coordi = points_data[:, 2]

#Formulating the matrix for linear regression
#The matrix A is created by combining the X and Y coordinates along with an␣
 ↪additional column of ones
A_matrix = np.column_stack([X_coordi, Y_coordi, np.ones_like(Y_coordi)])
# Vector B containing the Z coordinates
B_vector = Z_coordi

# Finding A transpose
A_t = np.transpose(A_matrix)

# This operation yields the left-hand side of the linear equation
A_t_times_A = np.matmul(A_t, A_matrix)

# This operation yields the right-hand side of the linear equation
A_t_times_B = np.matmul(A_t, B_vector)

# This is required to solve the linear equation
inv_A_t_times_A = np.linalg.inv(A_t_times_A)

# This gives the coefficients of the plane equation
coef = np.matmul(inv_A_t_times_A, A_t_times_B)

# Extract coefficients for the plane equation
a, b, c = coef

print("The obtained coefficients are: ", a, b, c)

# Meshgrid generation for plotting purposes
```

```
x_plane = np.linspace(X_coordi.min(), X_coordi.max(), 15)
y_plane = np.linspace(Y_coordi.min(), Y_coordi.max(), 15)
x_plane, y_plane = np.meshgrid(x_plane, y_plane)
z_plane = a * x_plane + b * y_plane + c

# Plotting surface and data points
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
# Plot the data points
scatter = ax.scatter(X_coordi, Y_coordi, Z_coordi, color='r')
# Plot the plane surface
plane = ax.plot_surface(x_plane, y_plane, z_plane, alpha=0.50)
# Set axis labels and title
ax.set_title('Obtained_plane')
ax.set_xlabel('X_axis')
ax.set_ylabel('Y_axis')
ax.set_zlabel('Z_axis')
plt.show()
```

```
The obtained coefficients are 0.577350269183371 3.7108171785364945e-12
-9.175141721101364e-12
```



7 Problem 4 (30 Points)

Link to the input image: https://drive.google.com/file/d/1VeZyrPIwyg7sqi_I6N5zBUJP9UHkyDEb/view?usp=sh

Given the photo of a train track, describe the process to find the mean distance in pixels between the train tracks for every row of the image where train tracks appear. This question contains two parts:

1. Design Pipeline and explain the requirement of each step

2. Write code to implement the pipeline

Problem-4 (1) To find the mean distance in pixels between the train tracks for every row of the image, the following pipeline can be designed and explained step by step:

Read the Image: - Use OpenCV (`cv2`) to read the image file. - Convert the image from BGR format to RGB format using `cv2.cvtColor`.

Track the Source and Destination Points for Warping: !!! - Run the loop to check the first pixel from left on the left track - In reverse direction run the loop again to check the first pixel from right on the right track - This will give us a trapezoidal map that needs to be warped into a rectangle to get the tracks parallel

Compute Perspective Transform Matrix: - Use `cv2.getPerspectiveTransform` to compute the perspective transform matrix (`M`) using the defined source and destination points.

Apply Perspective Transform: - Use `cv2.warpPerspective` to apply the perspective transform to the image using the computed transform matrix (`M`).

Convert Image Channels to HSV: - Convert the warped image from RGB to HSV (Hue, Saturation, Value) color space using `cv2.cvtColor`.

Blur the Image: - Apply median blur (`cv2.medianBlur`) to the HSV image to smooth out noise.

Separate HSB Channels: - Split the blurred HSV image into individual channels for Hue (`h`), Saturation (`s`), and Value (`v`).

Find Track Locations: - Iterate through each row of the Value channel (`v`). - Use `np.where` to find indices where the pixel value is 255 (indicating the presence of tracks) in each row. - Store these indices in `indices_255` for further processing.

Calculate Track Distances: - Iterate through the indices of rows where tracks are present (`indices_255`). - Find the maximum and minimum indices in each row to calculate the distance between tracks (`track_dist`). - Store these distances in `track_dist_list`.

Compute Mean Distance: - Calculate the mean distance in pixels between tracks by taking the average of all distances in `track_dist_list`. - Display or store the computed mean distance for further analysis.

This pipeline involves image preprocessing, perspective transformation, channel separation, track detection, and distance calculation to achieve the goal of finding the mean distance between train tracks in the image for each row where tracks are present.

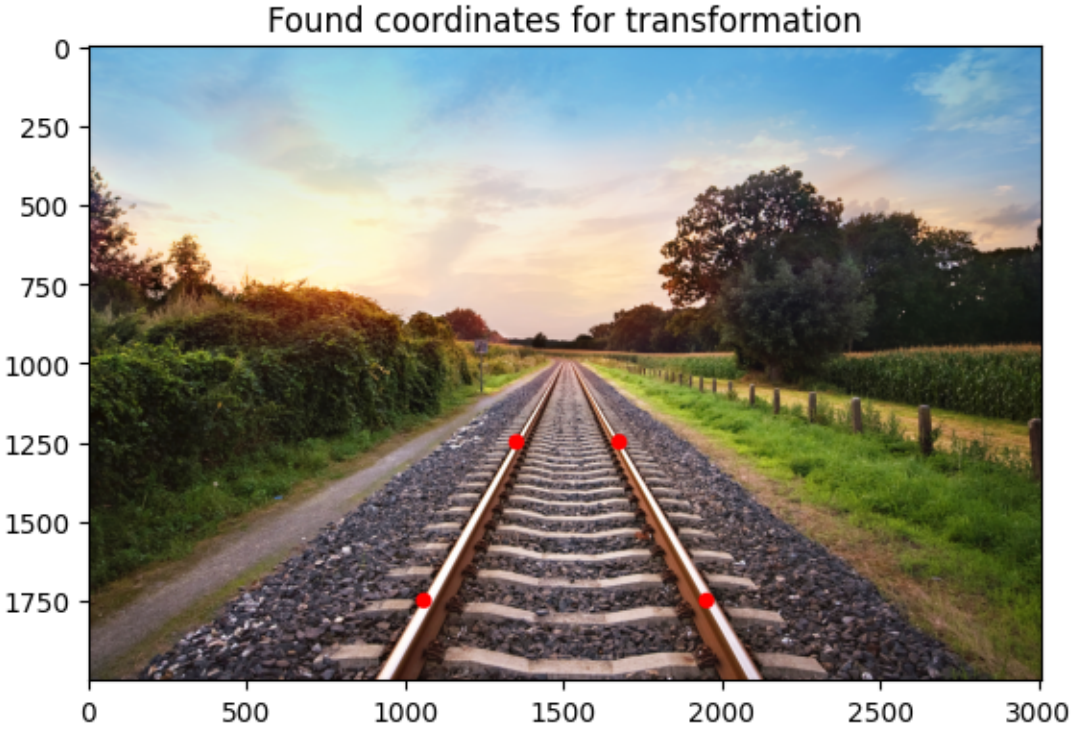
Problem-4 (2)

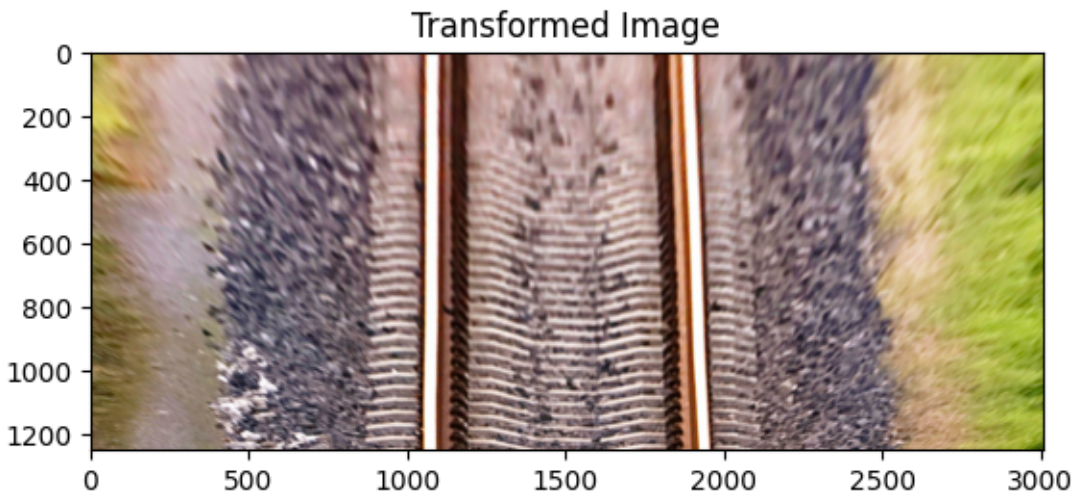
[85]:
```
# Read the image
img_bgr = cv2.imread('assets/train_track.jpg')
img_tr = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)
img_tr1 = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)
plt.imshow(img_tr)
plt.title('Given Image')
plt.show()

# Converting into gray image to be able to set threshold value >=250
img_gray = cv2.cvtColor(img_tr, cv2.COLOR_BGR2GRAY)

# Estimated values from horizon where the tracks start
y1 = y2 = 1250
y3 = y4 = 1750

# Loops to find the brightest pixel on track for the specified row
for i in range(len(img_gray[0])):
    pixel_value = img_gray[y1][i]
    if pixel_value >= 250:
        x1 =i
        break
# Printing circle at found pixel
cv2.circle(img_tr, (int(round(x1)) , y1), 25, (255 , 0 , 0 ), -1)
plt.imshow(img_tr)

# Loops to find the brightest pixel on track for the specified row
for i in range(len(img_gray[0])-1, -1, -1):
    pixel_value = img_gray[y2][i]
    if pixel_value >= 250:
        x2 =i
        break
# Printing circle at found pixel
cv2.circle(img_tr, (int(round(x2)) , y2), 25, (255 , 0 , 0 ), -1)
plt.imshow(img_tr)

# Loops to find the brightest pixel on track for the specified row
for i in range (len(img_gray[0])):
    pixel_value = img_gray[y3][i]
    if pixel_value >= 250:
        x3 = i
        break
# Printing circle at found pixel
cv2.circle(img_tr, (int(round(x3)) , y3), 25, (255 , 0 , 0 ), -1)
plt.imshow(img_tr)

# Loops to find the brightest pixel on track for the specified row
```

```
for i in range (len(img_gray[0])-1, -1, -1):
    pixel_value = img_gray[y4][i]
    if pixel_value >= 250:
        x4 = i
        break
# Printing circle at found pixel
cv2.circle(img_tr, (int(round(x4)) , y4), 25, (255 , 0 , 0 ), -1)
plt.imshow(img_tr)
plt.title('Found coordinates for transformation')
plt.show()

# Assigning the calculated x-coordinates of rectangle for Warping
x1_new = x3
x2_new = x4
x3_new = x3
x4_new = x4
# Assigning the calculated x-coordinates of rectangle for Warping
y1_new = y1
y2_new = y2
y3_new = y3
y4_new = y4


# Define the source and destination points for perspective transformation
src_points = np.float32([[x1, y1], [x2, y2], [x3, y3], [x4, y4]])
dst_points = np.float32([[x1_new, y1_new], [x2_new, y2_new], [x3_new, y3_new],␣
 ↪[x4_new, y4_new]])

# Compute the perspective transform matrix
M = cv2.getPerspectiveTransform(src_points, dst_points)

# Apply the perspective transform to the image
result_img = cv2.warpPerspective(img_tr1, M, (3008, 2000))  # Specify the␣
 ↪output image dimensions (width, height)

img_crp = result_img[0:y2, :]

# plt.subplot(1, 2, 2)
plt.imshow(img_crp)
plt.title('Transformed Image')
plt.show()


# Converting the image channels into HSV (Hue, Saturation, Brightness)
img_hsv = cv2.cvtColor(img_crp, cv2.COLOR_BGR2HSV)

#Blurring the image for a smoothing out the extra noise
```

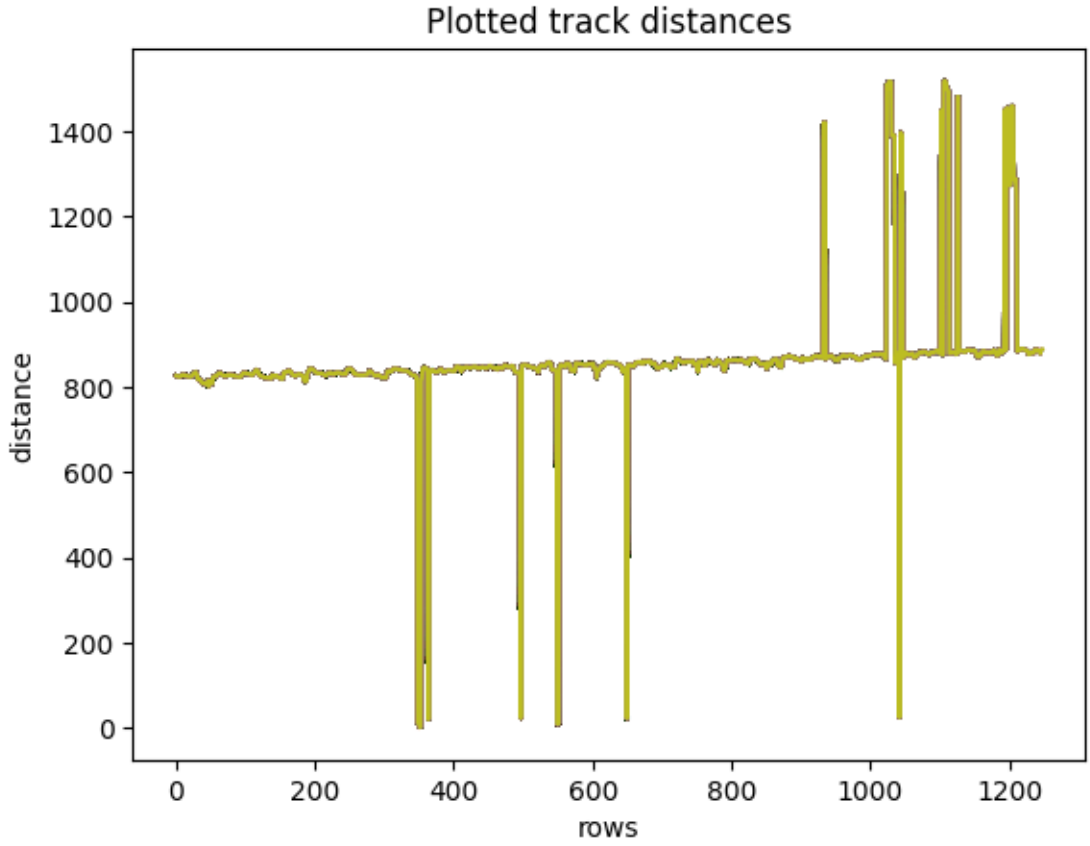
```
img_med = cv2.medianBlur(img_hsv, 5)

#Separating away HSB channels
h,s,v = cv2.split(img_med)

# Initialize lists to store max value and its index for each row
max_value_indices = []
max_values = []
min_value_indices = []
min_values = []
indices_255 = []
track_dist_list = []

# Iterate through each row
for row in v:
    # Find indices where the value is 255 using np.where
    indices = np.where(row == 255)[0]  # [0] to get the indices array from the␣
 ↪tuple returned by np.where

    # Append the indices to the list if the array is not empty
    if len(indices) > 0:
        indices_255.append(indices)



#Finding the difference between the track and storing the differences of the␣
 ↪each row
for row in indices_255:
    max = np.max(row)
    min = np.min(row)
    track_dist = max -min
    track_dist_list.append(track_dist)
    plt.plot(track_dist_list)
    plt.title('Plotted track distances')
    plt.xlabel('rows')
    plt.ylabel('distance')


#Finding the mean of all the distances in the respective rows
sum = 0
for i in track_dist_list:
    sum += i
mean = sum/np.size(track_dist_list)
print("\n mean distance in pixels: ", mean, "\n")
```


Given Image
0
250
500
750
1000
1250
1500
1750
0
500
1000
1500
2000
2500
3000


Found coordinates for transformation
0
250
500
750
1000
1250
1500
1750
0
500
1000
1500
2000
2500
3000




```
mean distance in pixels:  863.2546036829464
```



8 Problem 5 (15 Points)

Let's say you want to design a system that will create a top view of the surrounding around the car, see the image below for referene. Do the following,

(1) Describe and point the location of the sensors on the car in the given picture. Also, show the field of view of the sensors.

(2) Write the pipeline that will take input from these sensors and output a top-view image similar to the one shown below.

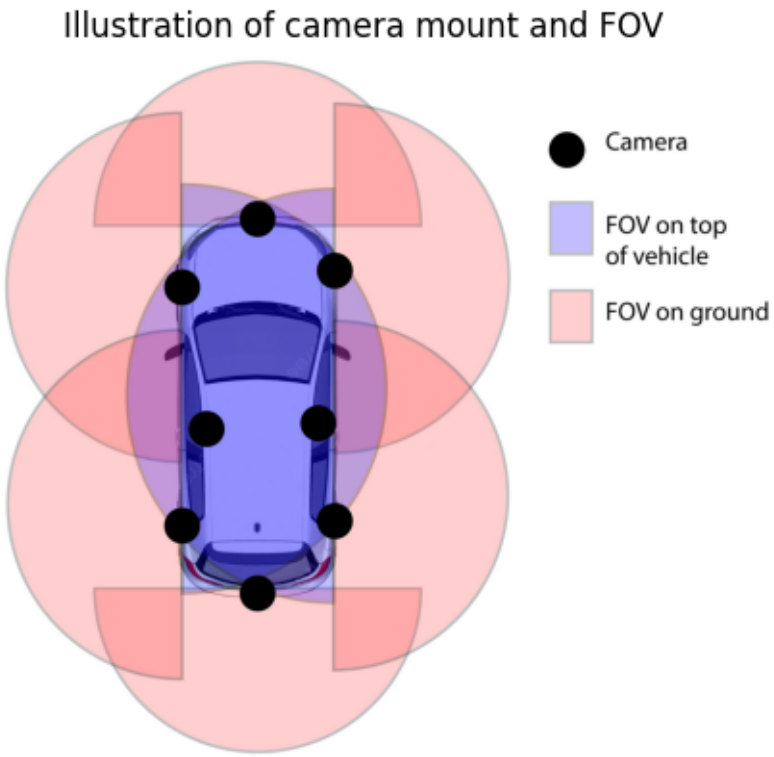
Note: Explain the answer in detail and provide necessary justifications for the steps you use in the pipeline.

Q5.1 > I have generated a picture below to show the possible positions of the different sensors/camera responsible for capturing the phootages of the surrounding ground of the car as well as the roof top for creating a complete top images of the vehicle along with its surrounding.

> Two cameras at the front and rear ends cover the either field of views (FOV)with 180 degrees sight angle. Two other pair of cameras on the either sides cover the lateral view with the similar FOV. Finally two cameras on the top for capturing the vehicle roof within the frame.All this helping in finding key points and stiching the images into one top view image.

[38]:
```
# Read the image
img_Q5= cv2.imread('assets/Q5.jpg')
img_Q5_RGB = cv2.cvtColor(img_Q5, cv2.COLOR_BGR2RGB)
plt.imshow(img_Q5_RGB)
plt.axis('off')
plt.title('Illustration of camera mount and FOV')

#BELOW IMAGE HAS BEEN CREATED BY 'ME' IN ILLUSTRATOR FOR PRESENTATION PURPOSE
 ↪==================
```

[38]:
```
Text(0.5, 1.0, 'Illustration of camera mount and FOV')
```



Q5.2

To create a top-view (panaromic) image from the above discrete images captured by seven cameras mounted as described (two at the front and rear ends covering 180-degree FOV, two on either side covering lateral views, and two on the top capturing the roof), the following image processing pipeline can be used:

Image Concatenation: - Load the seven discrete images from the respective file paths using OpenCV (e.g., image_path1 to image_path7) and concatenate them horizontally using `cv2.hconcat` to form a single row of images on an empty canvas.

Feature Detection and Matching: - Convert the images to grayscale and use the SIFT (Scale-Invariant Feature Transform) detector to detect keypoints and extract descriptors for each image. - Match the keypoints between adjacent images using the Brute-Force Matcher (BFMatcher) and filter the matches based on a distance ratio threshold.

Homography Estimation: - Calculate the homography matrix using RANSAC (Random Sample Consensus) based on the matched keypoints to find the transformation between image pairs.

Image Stitching (can be done using cv2 function or mannually): - Implement an image stitching function (`image_stitcher`) that takes two images as input and performs the following steps: - Detect keypoints and descriptors for the input images. - Filter keypoints and matches using the BFMatcher. - Find the homography matrix using RANSAC. - Warp the first image onto the base image using the homography. - Combine

the warped image with the second image to stitch them together.

Iterative Stitching: - Stitch pairs of images iteratively starting from adjacent images (e.g., image_path1 and image_path2) and continuing until all images are stitched together. - Crop the stitched images to remove any black areas or artifacts caused by the stitching process.

Display and Save Result: - Display the final stitched panoramic image using `cv2_imshow` and save it if needed. - Close all OpenCV windows to release resources using `cv2.destroyAllWindows()`.

By following this pipeline, the discrete images captured by the seven cameras can be seamlessly stitched together to create a panoramic top view image of the vehicle along with its surrounding environment, facilitating key point detection and image stitching for a comprehensive view.